Concurrency in databases

Priit Järv, Tallinn University of Technology

2014

1 Introduction

These lecture notes give an overview of how concurrency control can be implemented in databases. This includes both locking and alternative methods. We will start with an introductory example in PostgreSQL to show why concurrency control matters and then will cover selected implementation topics. The bibliography at the end has plenty of material that covers the subject in more breadth and detail.

2 Basics

When discussing concurrency, the central concept is the transaction. What is that? Imagine that there are two bank customers. One of them decides to transfer 50 euros from their account numbered 11314 to the other who's account number is 7735. So some program in the bowels of the gigantic bank infrastructure starts working and runs `update_acct(11314, -5000)`. Then it goes on to run `update_acct(7735, 5000)`, but that function returns with some I/O error. Turns out the server room air conditioner was poorly maintained, the RAID array overheated and went offline. Or something like that. Anyway, it seems that the bank has come out with at least something positive - they made 50 euros out of nothing!

Since banks usually don't want to make money like this (they have other ways), it was realized quite early in the history of databases that the logical unit of a transaction is needed. It would group together multiple changes to the database, and those changes only make sense if they are *each* completed, or else none at all.

To do that, after the program finishes all of its changes to the database, it issues either a "commit" command, which makes the changes permanent, or takes them back. Because the dominant database programming language SQL has the `ROLLBACK` statement for this, we will call this action rollback. It is also frequently called "abort".

In our imaginary banking example, the program never had a chance to issue a commit, because the disks were offline. Once they're plugged back in, a proper

database system would start recovery, realize that the change to the account 11314 is uncommitted and would undo that change.

3 Concurrent transactions

Try this in PostgreSQL (example done in version 9.1). Preparation:

```
=> create table seats (num integer,
->   booked char(1) default 'N',
->   cust_id integer);
CREATE TABLE
=> insert into seats (num) values (44);
INSERT 0 1
```

This creates a very simplified bookings database, where we have numbered seats and a flag that says whether the seat is already booked or not.

Say we need to create a program such that if the customer presses a button on a web form, a free seat will be booked for that customer. Here parameters are given in `psycopg` syntax, but that's just an example - we're not actually implementing this web application, so any syntax would do.

```
begin transaction;
select num from seats where booked='N';
update seats
  set booked='Y', cust_id=%(c_id)s
  where num=%(num)s;
commit;
```

So, first we check which seats are not booked yet and if any are available, we book the first one for the customer logged in on the website. We're doing this properly in a transaction, so that we can roll back all changes in case there is an error - the database driver will usually manage the transactions, but this is spelled out in SQL for clarity here. What could possibly go wrong?

Let's try it out in parallel. Open two windows with `psql` running. We'll be going in slow motion, statement by statement, so that we can see what is happening. In reality this will be much faster, but this specific order of actions could still occur:

window 1	window 2
`=> begin transaction;`	
`BEGIN`	
	`=> begin transaction;`
	`BEGIN`
`=> select num from seats`	
`-> where booked='N';`	
`num`	
`-----`	
` 44`	
`(1 row)`	
`=> update seats`	
`-> set booked='Y', cust_id=1`	
`-> where num=44;`	
`UPDATE 1`	
	`=> select num from seats`
	`-> where booked='N';`
	`num`
	`-----`
	` 44`
	`(1 row)`
	`=> update seats`
	`-> set booked='Y', cust_id=2`
	`-> where num=44;`
	– this will block
`=> commit;`	`UPDATE 1`
`COMMIT`	
	`=> commit;`
	`COMMIT`

Both transactions succeeded. But this doesn't look right - we only had one seat in the first place, how come two customers were able to book it then? If we look at the table contents, we'll see that customer 2 "officially" got the seat.

We also noted that the update statement in window 2 blocks until the other transaction has commited. So the database engine actually can see that the two transactions are related, it just does not do anything to prevent our overbooking mishap.

How should the database engine handle the cases where concurrent transactions interfere with each other? People have thought of this before, and the SQL-92 standard formalized this in four different levels:

- `READ UNCOMMITTED` - transactions can see modifications that other transactions are doing, even if they haven't been committed yet (this is called "dirty read", more on that later).
- `READ COMMITTED` - data returned by `SELECT` statements can change within one transaction, but uncommitted data is not visible.

- `REPEATABLE READ` - the records returned by the `SELECT` statement are not allowed to change, if the statement is repeated. New records may appear ("phantom read").
- `SERIALIZABLE` - the result of the transactions is equivalent to what would happen if the transactions would be executed serially (i.e. not concurrently). This also implicitly prevents the "phantom read" phenomenon - if there is only one transaction running concurrently, new records cannot appear from anywhere.

It turns out that the default level in PostgreSQL is actually `READ COMMITTED`[1]. So all we're guaranteed by the server, if we're to go by the above standard definition, is that we do not see uncommitted data. Was this even useful? In our experiment, if we had seen that the seat is already booked in window 2, overbooking wouldn't have happened, right? However, this is not the way to go[2]. It is not correct to assume that all the transactions always commit successfully. If, in our example, the transaction in window 2 would see that the seat is booked, but the transaction in window 1 would then encounter some error and roll back, the opposite of what happened in our experiment would occur - neither customer can book the seat!

Then we might want to look the other way and go with the strictest level possible - the `SERIALIZABLE` level. And we would not be wrong:

[1] `http://www.postgresql.org/docs/current/static/transaction-iso.html`

[2] In this particular case, PostgreSQL does not even implement `READ UNCOMMITTED`. There are SQL engines that do, however.

window 1	window 2
=> begin transaction -> isolation level serializable; BEGIN	
	=> begin transaction -> isolation level serializable; BEGIN
=> select num from seats -> where booked='N'; num ----- 44 (1 row)	
=> update seats -> set booked='Y', cust_id=1 -> where num=44; UPDATE 1	
	=> select num from seats -> where booked='N'; num ----- 44 (1 row) => update seats -> set booked='Y', cust_id=2 -> where num=44; ERROR: could not serialize access due to concurrent update
=> commit; COMMIT	
	=> rollback; ROLLBACK

We only need to exercise caution so that we do not use the results of the queries outside of the transaction until it has succeeded. In this case, even though in window 2 seat number 44 is returned, it shouldn't be displayed to the customer.

There is another and probably much more common method which does not rely on enforcing serializable schedules (for more information about serializability and its theoretical aspects, see [2] and especially [4]). Rather, the rows can be explicitly locked if we know that we're going to do something with them.

This can be achieved by the `FOR UPDATE` clause. In PostgreSQL we can rewrite our program as follows:

```
begin transaction;
select num from seats where booked='N' for update;
```

```
update seats
  set booked='Y', cust_id=%(c_id)s
  where num=%(num)s;
commit;
```

In this case, when the transaction executes, all rows where booked='N' are explicitly locked, so they cannot be modified until the transaction ends by commiting or rolling back. This restriction includes other `FOR UPDATE` queries as well. In other databases, you may need to declare a cursor to use this locking approach.

4 Anomalies and conflicts

What kind of problems and misfortunes could arise from concurrent access? This question is quite important to the database implementer[3] and it might be enlightening to go through before we move on to discussing implementation.

First, lets introduce some notation, so that we can describe the transactions concisely. $T1$, $T2$ and so forth denote transactions. A transaction itself consists of operations, for example: $r[x]$, $w[x]$, c means that the transaction first reads data item x, then writes it and commits. Remember that from the outside perspective, the transaction must appear atomic, i.e. all of the operations happen or none at all. To describe a concurrent execution of several transactions (a "history" or a "schedule"), we write it so: $w1[x]$, $r2[x]$, $a1$, $w2[x]$, $c2$ meaning that first $T1$ writes x, then $T2$ also executes and reads x, then $T1$ aborts, then $T2$ writes x and commits.

We have to be a bit superficial with the coverage here, since these behaviours are closely connected with how the database has implemented concurrent access - which we haven't studied yet.

1. Dirty write [1] - $w1[x]$, $w2[x]$. Notice that the first transaction has neither committed nor rolled back at the time the second one writes the data item. Why is this bad? If we assume that the database really has just one copy of each data item, this makes it difficult to roll back a transaction - if $T1$ aborts, should it restore the value of x to what it was before? To know the answer, we would need to keep track of all the changes to x after $T1$ has modified it *and* whether those transactions committed or not.

2. Inconsistent retrieval [2][4] - $w2[x]$, $r1[x]$, $w2[x]$. One transaction reads partial results from another transaction that updates values; also called WR conflict and "dirty read".

3. Lost update [2][4] - $r2[x]$, $w1[x]$, $w2[x]$. This is what happened in the introductory example.

4. Unrepeatable read [4] - $r1[x]$, $w2[x]$, $r1[x]$; also called RW conflict.

[3] Avoiding the term "database developer" here, as this could be easily misunderstood to be the end user issuing SQL commands to the database system, in modern usage.

5. Phantom read. If a query contains a range, such as `WHERE AGE>23`, then new rows that match the predicate may be inserted while a transaction is running.

6. Read skew [1]. Now we come to more complicated interactions, where the errors are constraint related. For example, if there is a constraint predicate $P(x, y)$, then $r1[x]$, $w2[x]$, $w2[y]$, $c2$, $r1[y]$ may cause the constraint predicate not to be true from the point of view of $T1$.

7. Write skew [1] - $r1[x]$, $r2[y]$, $w1[y]$, $w2[x]$. Both transactions write a new version of one of the pair that is consistent with the old version but the new versions together might not satisfy the constraint predicate.

8. Missing or multiple rows. This is an odd beast that doesn't quite fit in with the others. Mostly it has to do with practical implementation and is caused by index inconsistency[4].

The nice thing about the `SERIALIZABLE` isolation level is that it prevents them all, by definition. Bernstein in his influential book remarks that it is the obvious approach to concurrency control, as it is easiest to understand for the user. However, apparently under the modern market pressure this belief has not stood the test of time, and major vendors offer `READ COMMITTED` as default instead.

What protection does that give us? Not that much, although they've included more goodies than the SQL-92 standard requires. By definition, inconsistent retrieval is ruled out, as that involves reading uncommitted data. The standard does not mention "dirty write" at all, practical implementations however prevent it at `READ COMMITTED` level. As we already saw, there is the possibility to select rows `FOR UPDATE`, which prevents lost updates. This is sometimes defined as a separate isolation level called "Cursor Stability", with slightly stronger isolation. In any case, this is what you get in both Oracle and PostgreSQL.

5 Implementing concurrency control: locking

There are two implementation approaches. One is locking, which says that if we want to avoid some conflict, we lock the resource involved so that it cannot be modified in an undesired way. The other is optimistic concurrency control that lets everyone modify everything, then check if there were any conflicts. In other words, the "optimistic" part is that we assume there usually are no conflicts. Conventional wisdom goes that locking is good when the probability that two transactions access the same data is relatively high, otherwise locking wastes more time and optimistic concurrency is the way to go. Both methods are in use in current mainstream database products.

[4] `http://technet.microsoft.com/en-us/library/ms190805%28v=sql.105%29.aspx`

There is a third and very obvious alternative - completely serialize access to the database. Mainstream relational database systems do not do this, as there is a large cost to be paid in terms of throughput - applications frequently access unrelated parts of the database which can be done in parallel. But before we go into more complicated solutions, we might look at this as a warming up exercise.

To implement serialized access, we can introduce something called a locking protocol. We call it a protocol, because it is a set of interaction rules the programs that execute the transactions should follow. We add two operations: l and u, lock and unlock, respectively. The protocol is as follows: 1.) l is prepended to each transaction; 2.) u is appended to each transaction; 3.) if the l operation succeeds, we say that the transaction issuing holds the lock; 4.) if one transaction holds the lock and another transaction attempts l, it will be delayed until the lock holder issues u. It should not be difficult to see that this is enough to completely serialize all transactions.

An immediate improvement to this protocol can be done by observing that not all transactions need to change the contents of the database. Consider this history: $r1[x]$, $r1[y]$, $r2[x]$, $r2[y]$, $r3[x]$, $r3[y]$. This is the serial execution $T1$, $T2$, $T3$. Does it make any difference if the history is instead $r3[x]$, $r2[x]$, $r1[x]$, $r2[y]$, $r3[y]$, $r1[y]$? The database state will remain unchanged, so obviously we can safely interleave read operations from different transactions. However, this does not apply to write operations: $r1[x]$, $w2[x]$ is different from $w2[x]$, $r1[x]$. So a transaction that writes something cannot be safely interleaved with other transactions and needs exclusive access to the database.

With the improved lock protocol, transactions that are read only use a "shared" (S) lock. This lock is granted even if other shared locks exist, so read only transactions can work together. Transactions that modify something need to use an "exclusive" (X) lock, which cannot coexist with any other lock. This table, called a compatibility matrix, shows how the lock modes interact (Y - lock granted, w - block and wait):

	S (shared)	X (exclusive)
S (shared)	Y	w
X (exclusive)	w	w

We're now armed with enough basic understanding to move on. If there are two transactions $T1$: $r1[x]$, $w1[x]$ and $T2$: $r2[y]$, $w2[y]$, executing the operations in any order - provided that the order inside the transaction is preserved - would have the exact same outcome, even though they contain writes. Neither transaction has anything to do with what the other is modifying in the database. The implication of this is that completely serializing the access is not always necessary and there may be big gains in allowing transactions to run in parallel, even if the transactions write to the database. The trick is to somehow make sure that the *effect* of the operations always remains the same as if they were indeed serial.

When we look at the anomaly-causing histories, one thing is apparent. The anomalies are caused by a write in one transaction and a write or read of the

same item in another transaction. So this is what we actually mean by a conflict. If a transaction wants to write a data item, this operation must be delayed until the other transaction is done with it.

Finally we're left with how to determine when a transaction is "done" with a data item. First, it seems that if we do not want to read uncommitted data, we should delay reading that data from another transaction until commit (or abort), for example: $r1[x]$, $w2[y]$, $r2[z]$, $c2$, $r1[y]$ - even if $T2$ is doing something unrelated before it commits, we want to delay reading y from $T1$ since we don't know yet if the change is permanent - perhaps $T2$ rolls back.

Similarly, if we read data and want to provide at least the `REPEATABLE READ` isolation level, we need to delay any outside writes to the same data until the transaction ends. Other transactions reading that data are perfectly acceptable.

Again, the S and X locks are the tool to implement these delays, except that they now operate on some data item without affecting the rest of the database. In histories we can use the notation $sl[x]$ and $xl[x]$ to denote data-level locks. To delay reads of freshly written data, we take an $xl[x]$ lock on it just before we write it. To delay writes to the data we're about to read, a $sl[x]$ is inserted before the read operation. Once the transaction acquires a lock, in both cases it will be held until the transaction either commits or aborts. Because all locks are released together, the shorthand $u[x]$ is enough to denote an unlock operation.

This relatively simple reasoning has given us what is known as the Strict Two-phase Locking protocol[5], or Strict 2PL. For a proper correctness proof of the algorithm, see [2] or [3]. In fact, depending on how we define the locked data item, it provides isolation all the way to the `SERIALIZABLE` level - in this case the lock needs to be for a range of values, a.k.a a range lock, to prevent phantom reads.

Let's look at how our introductory example would behave with the concurrency implementation we've described:

[5]The nomenclature varies somewhat between authors, this is what Bernstein et al. use.

T1	T2
`=> begin transaction;`	
– nothing to do	
	`=> begin transaction;`
	– nothing to do
`=> select num from seats`	
`-> where booked='N';`	
$sl1[seats(num = 44)]$	
$r1[seats(num = 44)]$	
`num`	
`-----`	
`44`	
`(1 row)`	
`=> update seats`	
`-> set booked='Y', cust_id=1`	
`-> where num=44;`	
$xl1[seats(num = 44)]$	
$w1[seats(num = 44)]$	
`UPDATE 1`	
	`=> select num from seats`
	`-> where booked='N';`
	$sl2[seats(num = 44)]$
`=> commit;`	
$u1[seats(booked = 44)]$	
`COMMIT`	$r2[seats(num = 44)]$
	`num`
	`-----`
	`(0 rows)`
	`=> rollback;`
	$u2[seats(num = 44)]$
	`ROLLBACK`

To avoid overcomplicating things, we didn't use range locks here. Nevertheless, the row that would match the predicate is locked and cannot be returned; once the lock is released it no longer matches the predicate and still cannot be returned.

This scheme can be relaxed, should we want to improve throughput by allowing more operations in parallel. Earlier we noted that to provide at least `READ COMMITTED` level, we should hold exclusive locks until the transaction ends. Shared locks, however, may in this case be released immediately after the read operation is complete [1].

What about the useful "Cursor Stability" level? One way would be to immediately take exclusive locks when the query contains the `FOR UPDATE` clause. This might be good enough, but to squeeze out more performance, it would be nice to start with a shared lock to allow more concurrency, then upgrade to an

exclusive lock when we're ready to write.

Turns out that this can introduce a nasty and very common deadlock, as in the history $sl1[x]$, $r1[x]$, $sl2[x]$, $r2[x]$, $xl1[x]$, $xl2[x]$. Either transaction is entitled to upgrade their lock, but neither is able to, since the other holds a shared lock too. For this purpose, another lock mode can be introduced: the update (U) lock [3]. This mode has an asymmetric compatibility matrix (top row is lock held, left colum lock wanted):

	S	X	U
S	Y	w	w
X	w	w	w
U	Y	w	w

The update lock is granted if a shared lock is held, but once a transaction holds an update lock, no additional locks will be granted. This lock will be requested when the `FOR UPDATE` clause is present and will be upgradeable to an exclusive lock, provided that all concurrent shared locks are released. Because two upgrade locks are incompatible, the deadlock scenario above can no longer occur.

6 A detour: granularity levels

So far, what is a "data item" has been left unspecified. A natural choice from the users' point of view would be locking a table row, which would represent a logical unit of a record. However, we might also use page locks, based on the physical representation of data in memory and on disk.

Whatever we choose, there is a tradeoff involved. If the units are small (fine granularity), more concurrency is possible - an access of one row does not prevent another transaction working with another row. On the other hand, if we wish to lock large amounts of data, a lot of work and a lot of space is needed to maintain the locks. With large units (coarse granularity), there is more contention for the same resources, but you'd need to maintain a smaller number of locks. Bernstein has given some curves on how granularity affects the throughput [2].

But perhaps there is a way to get the best of both worlds - if we can, we only lock small units, but fall back to locking larger units whenever necessary. For example, we could use four levels: row, page, table and database locks. All we need to do is to figure out what happens if there are locks on different levels. Can we grant a lock on a table, if there is an exclusive lock on a table row? No, because the holder of the row lock assumes that no other transaction will interfere with whatever they are doing.

Fortunately, the structure of the database makes it relatively easy to manage locks on different levels. A database consists of tables, each with the same parent and a table similarly consists of rows. How pages and rows overlap isn't

so clear, but we can certainly implement the pages so that the this tree structure is preserved.

Then it is quite easy to see that if we hold an X lock on a table, no locks on page or row level can be granted. The other way around is only slightly more complicated. We introduce a new lock mode called an "intention lock". In its simplest form, the hierarchical protocol works like this: 1.) if we want to take an S lock, we first need an IS lock on the parent and similarly, an X lock requires an IX lock on a parent; 2.) IS lock requires an IS lock on the parent and IX lock requires an IX lock on the parent. This is enough to prevent any conflicts, given the following compatibility matrix:

	IS	IX	S	X
IS	Y	Y	Y	w
IX	Y	Y	w	w
S	Y	w	Y	w
X	w	w	w	w

The hierarchical protocol starts with the root: to get a row lock, the transaction must acquire an intention lock at the database level, then move down the hierarchy, getting intention locks along the way until reaching the row level.

Intention locks are compatible with each other, so the transactions at row level can peacefully co-exist most of the time, unless they happen to hit the same rows. IS lock is also compatible with S lock, meaning that locking, for example, the entire table for reading does not interfere with transactions that lock individual rows for reading.

Note that the higher we go up the hierarchy, the harder it becomes to lock the entire structure: for example, a single X lock on a row would prevent getting an S lock on the database, as according to the protocol there is an IX lock present at database level. Also, new IX locks can come in at any time, while the S lock request has to wait[6].

7 Optimistic concurrency control: MVCC

Multiversion concurrency control (MVCC) is currently the most successful optimistic concurrency control method. As a matter of fact, it is even the leader in commercial RDBMS market, thanks to Oracle. There are multiple versions described in the literature, the textbook by Garcia-Molina et al. [3] gives a practical one that also supports rollbacks of transactions.

Each data item has multiple copies, denoted x_i where i is the version. Each copy has the following metadata: read timestamp $RT(x_i)$, write timestamp $WT(x_i)$ and a commit bit $C(x_i)$. The commit bit prevents reading data from transactions that later roll back.

[6]This can be remedied with fair scheduling, but there is no single dominant solution - a strict FIFO queue could improve average latency of individual transactions, but would at the same time reduce throughput in read-heavy settings.

The transactions also receive a timestamp when they begin, $TS(T)$. Each transaction still consists of r, w, c and a operations, as before.

Algorithm 1 MVCC: Read operation

```
1: Find the newest x_i such that WT(x_i) ≤ TS(T)
2: if C(x_i) then                                          ▷ committed
3:     Read x_i
4:     RT(x_i) ← MAX((TS(T), RT(x_i))
5: else
6:     Wait until C(x_i) or the transaction with TS(T) = WT(x_i) aborts
7: end if
```

The read operation (Algorithm 1) goes to the latest version of x and checks the write timestamp. If this is in the future, it will go down the chain to find a copy with a timestamp that is in the past. Note that the copy is guaranteed to be found. Why? Because $TS(T)$ was the highest timestamp in existence at the time the transaction started and we only initiate this procedure if the data item actually exists. This means that its history before $TS(T)$ also exists - this is ensured by the procedure for history purge (Algorithm 5).

Algorithm 2 MVCC: Write operation

```
1: Find the newest x_i such that WT(x_i) ≤ TS(T)
2: if TS(T) ≥ RT(x_i) then
3:     if WT(x_i) = TS(T) then
4:         Write x_i        ▷ overwrite, old data belongs to the same transaction
5:     else
6:         Write x_j                                       ▷ new version
7:         WT(x_j) ← TS(T)
8:         Unset C(x_j)
9:     end if
10: else
11:     Restart with new TS(T)                          ▷ cannot change the past!
12: end if
```

The write operation (Algorithm 2) also finds the copy that is in its snapshot, but only to check the read timestamp. The restart happens if $WT(x_i) < TS(T) < RT(x_i)$. This means that some committed transaction has written the value and another transaction has already read it. If we write our value now that the read with a newer timestamp has already completed, we get a situation that cannot happen if the transactions execute serially in timestamp order.

Algorithm 3 MVCC: Commit

1: **for all** x_i where $WT(x_i) = TS(T)$ **do**
2: Set $C(x_i)$
3: Unblock transactions waiting on $C(x_i)$
4: **end for**

Algorithm 4 MVCC: Rollback

1: **for all** x_i where $WT(x_i) = TS(T)$ **do**
2: Delete x_i
3: Restart the read operation of transactions waiting on $C(x_i)$
4: **end for**

Algorithm 5 MVCC: History purge

1: Find the newest x_i such that $WT(x_i) \leq \min\{TS(V) \mid V \text{ is active}\}$
2: **for all** x_j where $j < i$ **do**
3: Delete x_j
4: **end for**

This approach works in that it produces serializable histories [6] and transactions can be safely rolled back, unfortunately there are some performance issues. First, the rollback (Algorithm 4) and restart of transactions is potentially expensive - various authors consider it more expensive than waits in locking [3][6]. Second, the recoverability protocol using the commit bit is a form of locking by itself and the readers are still blocked by writers through this mechanism.

Several popular RDBMS systems use an algorithm that is actually a combination of MVCC and 2PL. The method is quite elegant, although it sacrifices some isolation for performance. The idea is to use the multiversion scheme for reads, so that each read can execute in their own virtual snapshot of the database. Writes are handled using Strict 2PL.

The metadata for each copy is simplified: data items in this algorithm have a single timestamp. The commit bit is no longer needed. There is also a global counter for timestamps TS_{curr}[7]. The transactions are divided between read-only transactions and update transactions, i.e. those that also issue write operations.

Read-only transactions only need to execute reads (Algorithm 6). Their commits and aborts are no-ops with this algorithm.

Algorithm 6 MVCC+2PL: Read operation

1: Find the newest x_i such that $TS(x_i) \leq TS(T)$
2: Read x_i

When a read-only transaction starts, its timestamp is assigned the value of

[7] Oracle calls this the "system change number" (SCN).

TS_{curr}. There is no need to increment TS_{curr}, because read-only transactions do not affect the state of the data; from the point of view of other transactions they are invisible. However, they do need the timestamp to extract their own snapshot view of the database.

Update transactions do a bit more work. They also get their initial timestamp from TS_{curr}, which is used for reading data, so the read operation is identical to read-only transactions. The write operation (Algorithm 7), however, requires locking. Note that the X lock is not released.

Algorithm 7 MVCC+2PL: Write operation

1: Get exclusive lock on x
2: $x_j \leftarrow$ new version of x
3: $TS(x_j) \leftarrow +\infty$ ▷ so that no one reads this yet

The commit operation (Algorithm 8) is where the magic happens. This is also the part of the algorithm which is potentially costly to implement, as some bookkeeping and synchronization is needed.

Algorithm 8 MVCC+2PL: Commit

1: Block other commits
2: **for all** x_i that the transaction created **do** ▷ need to keep a list
3: **if** $\exists x_j, TS(T) < TS(x_j) < TS_{curr} + 1$ **then**
4: Abort with a write-conflict error ▷ "first committer wins" protocol
5: **end if**
6: $TS(x_i) \leftarrow TS_{curr} + 1$
7: **end for**
8: $TS_{curr} \leftarrow TS_{curr} + 1$ ▷ the new values become visible atomically
9: Unblock other commits
10: Release all X locks held

Rollback (Algorithm 9) is again greatly simplified. The purge procedure works similarly to the "pure" MVCC algorithm described earlier.

Algorithm 9 MVCC+2PL: Rollback

1: Release all X locks held
2: **for all** x_i that the transaction created **do**
3: $TS(x_i) \leftarrow -\infty$ ▷ or just delete it immediately
4: **end for**

Returning once more to the commit procedure: the commit timestamp check is there to prevent the lost update anomaly. If it is implemented, the algorithm provides what is called "Snapshot Isolation", which we haven't mentioned before. It is considered to be close to the `REPEATABLE READ` level, with some differences regarding which anomalies are allowed to sneak through [1].

This algorithm is also roughly what PostgreSQL provided before version 9.1 as it's strongest isolation level [5] and should also match Oracle's `SERIALIZABLE` level[8]. We can weaken it by removing the "first committer wins" check. If we also assign a new timestamp after each SQL statement, we get the equivalent to the `READ COMMITTED` level of PostgreSQL and Oracle [5].

8 Who uses what?

Locking was the first mainstream method and the historical System R from IBM and INGRES (ancestor of Postgres) from UC Berkeley implemented Strict 2PL. Optimistic concurrency control was seen at the end of 1970's to be the promising alternative, and InterBase and Oracle, of the commercial systems currently still in wider use, claim to be among the pioneers in the first half of the 1980's.

We've mostly covered Oracle and PostgreSQL above, further online documentation is available at `http://docs.oracle.com/cd/B14117_01/server.101/b10743/consist.htm` and `http://www.postgresql.org/docs/current/static/transaction-iso.html`.

The locking approach is still alive and well among the RDBMS. IBM DB2 uses Strict 2PL where X and U (update) locks are held until end of transaction. Two granularity levels are provided: table and row. There are in total 10 lock modes, some of them used to implement range locks. All four canonical SQL-92 isolation levels are provided. [6]

MS SQL server offers both locking and optimistic concurrency. Strict 2PL with range locks uses S, U, X modes and intent modes and 6 levels of granularity. Using locking, all SQL-92 isolation levels are provided [6]. Optimistic concurrency control was added more recently and seems similar to MVCC-based snapshot isolation that other vendors offer, with the relaxed `READ COMMITTED` level available too.

MySQL has two storage engines: MyISAM, which does not support transactions, but uses table level locking for each statement; and InnoDB, which is rather similar to Oracle and PostgreSQL, except that its default isolation level is `REPEATABLE READ` and it provides all SQL-92 levels.

9 Final remarks

It would be an exaggeration to say that what we've covered here is just the tip of the iceberg. What we've covered is the core of concurrency control, but there are several complications that practical implementations need to deal with. We made a side trip to visit one of them, the multi-granular locking, as it is intuitive and generally instructive.

There is, for example, the question of deadlock prevention. Both pure Strict 2PL and the multiversion concurrency scheme we've described are vulnerable to

[8] This is what various documentation and literature hints at; the exact details are probably a trade secret.

deadlocks. Consider the following, perfectly legal history: $xl1[x]$, $w1[x], xl2[y]$, $w2[y]$, $xl1[y]$, $xl2[x]$ and we have a deadlock, as neither $T1$ or $T2$ will release their locks, thus neither of them will proceed.

An important part of a database is the index. It, too, requires concurrency control, as different transactions want to read and write it concurrently. Should we serialize the access to the index, we would lose all of the advantage we've gained by implementing the complex schemes for controlling access to the data items. The index is most commonly a tree structure, and an update to a data item can affect parts of the index that are related to data the query itself is not touching. A sound approach to synchronizing index access and data access is therefore needed.

Recovery is usually closely connected to concurrency control. We've only mentioned the need for it and pointed out some complications in algorithms which were introduced to support recoverability.

To keep the scope of these notes reasonable, we will not go into further detail about those subjects, but rather direct the reader to the several textbooks in the bibliography section.

As we could see, SQL standards and mainstream implementations are quite different in their approach. The textbooks also diverge - usually being correct, but for readability purposes glossing over some practical aspects or being in danger of becoming somewhat outdated in their more opinionated parts - the former being the case with newer books, while the latter especially applies to the still influential texts by Bernstein and Gray which are now over two decades old.

It is perhaps a good time to point out that concurrency control does not just concern the SQL databases. The isolation levels described earlier are specifically SQL standard, but the anomalies and conflicts can occur anywhere there is concurrent reading and writing of shared data records.

Bibliography

[1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, *A critique of ansi sql isolation levels*, ACM SIGMOD Record, vol. 24, ACM, 1995, pp. 1–10.

[2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency control and recovery in database systems*, vol. 370, Addison-Wesley New York, 1987.

[3] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom, *Database systems: The complete book*, Pearson Education, 2002.

[4] Jim Gray and Andreas Reuter, *Transaction processing*, Morgan Kaufmann Publishers, 1993.

[5] Dan RK Ports and Kevin Grittner, *Serializable snapshot isolation in postgresql*, Proceedings of the VLDB Endowment **5** (2012), no. 12, 1850–1861.

[6] Abraham Silberschatz, Henry F Korth, and S Sudarshan, *Database system concepts*, McGraw-Hill New York, 2002.